An Efficient Algorithm for Node-Weighted Tree Partitioning with Subtrees’ Weights in a Given Range

Guangchun LUO†, Hao CHEN‡†, Members, Caihui QU†, Yuhai LIU††, and Ke QIN†, Nonmembers

SUMMARY Tree partitioning arises in many parallel and distributed computing applications and storage systems. Some operator scheduling problems need to partition a tree into a number of vertex-disjoint subtrees such that some constraints are satisfied and some criteria are optimized. Given a tree \( T \) with each vertex or node assigned a nonnegative integer weight, two nonnegative integers \( l \) and \( u \) \((l < u)\) and a positive integer \( p \), we consider the following tree partitioning problems: partitioning \( T \) into minimum number of subtrees or \( p \) subtrees, with the condition that the sum of node weights in each subtree is at most \( u \) and at least \( l \). To solve the two problems, we provide a fast polynomial-time algorithm, including a pre-processing method and another bottom-up scheme with dynamic programming. With experimental studies, we show that our algorithm outperforms another prior algorithm presented by Ito et al. greatly.

key words: tree partition, operator scheduling, dynamic programming, distributed computing

1. Introduction

Pipelined operator scheduling is an important problem in the area of parallel and distributed computing \([1, 2]\). For a large class of applications, the computing tasks can be represented as pipelined operator graphs, and an efficient operator schedule is critical to reach the best possible performance for distributed computing. Here we investigate a special class of operator scheduling problems for trees. Consider an arbitrary tree \( T \) with nonnegative integer weights on the nodes. Each node \( v \) in \( T \) is called an operator and assigned a weight \( w(v) \). We aim to partition \( T \) into disjoint \( p \) clusters by deleting edges from \( T \) such that each cluster is a subtree and the total weight of each subtree is in a given range \([l, u]\), where \( l \) and \( u \) are two nonnegative integers and \( l < u \). Such a partition is called a \( p-[l, u] \)-partition of \( T \).

In this paper, we mainly deal with the following problem: partitioning \( T \) into minimum number of node-disjoint subtrees satisfying that the total weight of each subtree is in the given range \([l, u]\), which is called minimum \([l, u]\)-partition problem. We present a polynomial-time algorithm to solve the problem. The concept of the algorithm is as follows. Firstly, we adopt a preprocessing approach to transform the original tree \( T \) into a number of new trees for which the weight of each leaf node is in \([l, u]\). Then we use a dynamic programing formulation to find a minimum \([l, u]\)-partition for each of these new trees, where a few pruning methods are presented to speed up the computing process. At last, we combine minimum \([l, u]\)-partitions for these new trees and get the minimum \([l, u]\)-partition for \( T \) as the final result. The proposed algorithm only needs to be modified slightly to solve the \( p-[l, u] \)-partition problem. A series of experiments are conducted to evaluate our algorithm and compare it with the state-of-the-art polynomial-time algorithm presented by Ito et al. \([3]\). Experimental results reveal that our proposed algorithm runs much faster than the algorithm presented by Ito et al. and reduce the total computing time greatly in different cases.

The paper is structured as follows. Section 2 contains a review of related works. In Sect. 3 we formalize the problem and present a pre-processing algorithm to simplify the original problem. Based on the pre-processing step, we then give a polynomial-time algorithm for the minimum \([l, u]\)-partition problem in Sect. 4. Section 5 presents a slightly modified algorithm to solve the \( p-[l, u] \)-partition problem. Section 6 evaluates our novel algorithm. Finally, Sect. 7 concludes this paper.

2. Related Work

Our work is mainly related to the research of pipelined operator scheduling. Pipelined operator tree (POT) scheduling is an important problem in the area of parallel query optimization. A schedule assigns \( n \) operators of a tree to \( p \) processors. Weights associated with nodes and edges of a tree represent respectively the cost of operators and communication between them. There are a few kinds of tree scheduling problem depending on different application fields and different optimization objectives. Some scheduling don’t consider the communication costs of processors, while others do. To evaluate the performance of a schedule, various optimization criteria are used.

The optimization target of some scheduling problem \([1, 4, 5]\) is to find a schedule with minimum response time (or makespan). The response time of a schedule is the maximum processor load. This kind of tree scheduling is NP-hard since even the special case with all communication costs being zero is a classical multiprocessor scheduling problem. S.E. Hambrusch \([6]\) considers partitioning the vertices of an \( n \)-vertex tree into \( p \) disjoint clusters so that the number of vertices in a cluster and the number of subtrees in a cluster are minimized. For this NP-hard problem,
S.E. Hambrusch presents greedy heuristics for k-clustering the vertices of a tree when cluster sizes need to lie in an upper limit defined by k. Kanne and Moerkotte [7] present a partitioning model called sibling partitioning, which considers the subtrees that are siblings as one part. They propose a linear time algorithm that optimally partitions an ordered, labeled, weighted tree such that each partition does not exceed a fixed weight limit. Jan-Jan Wu [8] proposes an optimal partitioning algorithm for metadata trees in the area of distributed storage systems, focusing on how to partition a hierarchical metadata structure into multiple metadata servers evenly. In [8], a dynamic programming approach is proposed in minimizing the workload imbalance among metadata servers. Ito et al. [3] present the [l, u] partition problem for trees and obtained a polynomial-time algorithm to solve p-[l, u] partition and minimum-[l, u] partition problems.

3. Problem Statement and Simplification

3.1 Terms and Definitions

Consider a tree \( T = (V, E) \) rooted with node \( r \), where \( V \) is a set of nodes and \( E \) is a set of edges. For any node \( v \) of \( T \), we denote \( v \)'s parent node by \( v.parent \) and denote the subtree of \( T \) which is rooted at \( v \), by \( T_v \). Each node of \( T \) is assigned a nonnegative integer weight. The weight of a subtree is defined as the sum of node weights in the subtree. Consider a tree \( T \) and subtree \( T_v \) is defined as the sum of node weights in the subtree. The weight of node \( v \) and subtree \( T_v \) are denoted by \( w(v) \) and \( w(T_v) \) respectively. The number of nodes for \( T \) is denoted by \( |T| \) or \( n (n = |T|) \). Assume we select \( k - 1 \) nodes from a subtree \( T_v \) and break \( T_v \) into \( k \) non-empty subtrees by cutting down the \( k - 1 \) edges which connect the \( k - 1 \) nodes with their parents. The set of the \( k - 1 \) nodes together with \( v \) is called a \( k \)-partition of \( T_v \), denoted by \( P^k_{T_v} \). Given a \( k \)-partition \( P^k_{T_v} \), we denote the set of partitioned subtrees by \( F(P^k_{T_v}) \) and call it partition forest. The weight of the subtree rooted with a node \( x \) in \( F(P^k_{T_v}) \) is denoted by \( W_x(P^k_{T_v}) \). The unique subtree which contains \( v \) for a \( P^k_{T_v} \) is called pivot subtree (pivot tree for short), and it’s weight is denoted by \( W_x(P^k_{T_v}) \).

Definition 1 (k-[l, u] partition). A k-partition for a tree \( T_v \) is a k-[l, u] partition, if the weight of each partitioned subtree is in \([l, u]\). Then, a k-[l, u] partition for \( T_v \) is called feasible.

There may exist many \( (J \) for example) k-[l, u] partitions for a tree \( T_v \), we can denote each of them by \( P^k_{T_v}(j) \), \( 1 \leq j \leq J \). \( P^k_{T_v}(j) \) is called leaest if the weight of pivot tree is minimum among all k-[l, u] partitions.

Definition 2 (minimum [l, u] partition). \( P^k_{T} \) is a minimum [l, u] partition if \( k \) is the minimum integer existed for k-[l, u] partition of a tree \( T \).

The problem to be solved in this paper is formally stated as follows: (1) Given a tree \( T \) with a weight range \([l, u]\), find a minimum [l, u] partition for \( T \). (2) Given a tree \( T \) with a weight range \([l, u]\) and a number \( p \), find a p-[l, u] partition for \( T \).

Before solving our problems, we present some properties which will be useful for optimizing our algorithm later.

Property 1. Assume p-[l, u] partition for a tree \( T \) is feasible. Consider a leaf node \( v \) of \( T \) with \( w(v) > l \). If we replace \( v \)'s weight with a new integer \( w' \), \( l \leq w' < w(v) \), \( T \)'s p-[l, u] partition is still feasible.

From property 1, we can easily deduce the following property.

Property 2. Assume p-[l, u] partition for tree \( T \) is feasible. Let \( T_v \) be a subtree of \( T \) and \( P^k_{T_v}(j) \) be a k-[l, u] partition of \( T_v \), \( k < p \). We suppose \( P^k_{T_v}(j) = \{x \in P^k_{T_v}(j') \mid x \in T_v\} \) holds for some \( P^k_{T_v}(j') \). If \( P^k_{T_v}(j) \) is not the leaest among all k-[l, u] partition of \( T_v \), then there must be another \( P^k_{T_v}(j'') \) for which \( P^k_{T_v}(j'') = \{x \in P^k_{T_v}(j'') \mid x \in T_v\} \) holds, where \( P^k_{T_v}(j) \) denotes the leaest k-[l, u] partition of \( T_v \).

3.2 Problem Simplification

Definition 3. A tree or subtree is called fitting if it’s total weight is in \([l, u]\). A node is called fitting if it’s weight is in \([l, u]\). A tree is called leaf-fitting tree (lf-tree for short) if all leaf nodes are fitting.

To reduce the scale of tree partitioning problems, we first translate an initial tree \( T \) into much smaller lf-trees. Now we present two properties that one can verify easily.

Property 3. Consider a leaf node \( v \) in tree \( T \) and suppose \( w(v) < l \). If \( w(v) + w(v.parent) > u \), then p-[l, u] partition of \( T \) doesn’t exist for any \( p \). If \( w(v) + w(v.parent) \leq u \), then \( v \) and its parent must belong to the same subtree for \( T \)'s any p-[l, u] partition.

Property 4. Let \( T \) be a lf-tree. Suppose there is a leaf node or an internal node \( v' \) of \( T \) and \( w(v') + w(v'.parent) > u \). If k-[l, u] partition \( P^k_{T} \) of \( T \) exists for any integer \( k \), then we have: \( v' \in P^k_{T} \).

Motivated by the properties above, we use a pre-processing algorithm (Algorithm 1) to glue together certain leaf nodes of \( T \) iteratively into their parent nodes with their weight adding together, and then partition some subtrees from the global tree, provided that doing so would not affect the final solution of our problems.

The pre-processing algorithm iteratively fuses pairs of nodes \((v.v.parent)\) for which the condition in property 3 holds (line 3, 14, 22, 23 of Algorithm 1). Once we fuse a node \( v \) into it’s parent, we update the parent’s weight as \( w(v.parent) = w(v[parent]) + w(v) \). After the fusing process, we turn the original \( T \) into a new tree \( T \) in which all leaf nodes are fitting. Then, we check each internal node’s children, and partition those nodes for which the condition in property 4 holds (line 10 to 11). By the iterative fusing and partitioning, we obtain at last a few new lf-trees, in which all leaf nodes are fitting and \( w(v) + w(v.parent) \leq u \) for any node \( v \). Figure 1 shows an example tree \( T_v \) and the lf-trees after pre-processing. Here we assume \( l = 5 \) and \( u = 14 \).

In Fig. 1, each node is represented as an oval with it’s identifier and weight. All fitting nodes are represented as a black oval. Clearly, nodes \( v_1, v_3, v_{22}, v_{23}, v_{41}, v_{43} \) are fused.
Algorithm 1 Pre-processing.

Require: 

T tree;

Ensure: 

TS(a set of If-trees);

1: for all nodes v of T in post order do 
2: if v is leaf then 
3: LeafProcess(v); 
4: else 
5: if w(v) > u then 
6: TS = φ; 
7: return; 
8: end if 
9: for any v’s child vj with w(vj) + w(vj) > u do 
10: cut off vj from T; 
11: add T_vj into TS; 
12: end for 
13: if v is leaf then 
14: LeafProcess(v); 
15: end if 
16: end if 
17: return TS; 
18: end for 
19: 
20: function: LeafProcess(v)
21: if w(v) < l then 
22: w(v.parent) = w(v.parent) + w(v); 
23: cut off v from T; 
24: end if 
25: if w(v) > u then 
26: TS = φ; 
27: return; 
28: end if 

into their parents, since they are not fitting leaf nodes. The edge (v, v_a) is deleted, since w(v) + w(v_a) > u. Our experiments in Sect. 6 show that this pre-processing step decreases the scale of problem greatly, and help to speed up the process of arriving at the final solution. The pre-processing algorithm has a runtime complexity of O(n).

4. Minimum [l, u]Partition Problem for Arbitrary Tree

4.1 Problem Analysis of p-[l, u]Partition for If-tree

To solve the p-[l, u]partition problem of a If-tree, we present some new definitions in advance.

Definition 4 (near k-[l, u]partition). A k-partition for tree T_v is called near k-[l, u]partition, denoted by P^k_v, if

W_v(\bar{P}^k_v) \leq u and l \leq W_v(\bar{P}^k_v) \leq u for each x ∈ \bar{P}^k_v \setminus \{v\}.

There may exist J near k-[l, u]partitions for a tree T_v, we can denote each of them by \bar{P}^k_v(j), 1 \leq j \leq J. A \bar{P}^k_v(j) is called fitting leanest if the weight of pivot tree for it is minimum among all near k-[l, u]partitions with pivot tree weighted in range [l, u]. According to definition 4, we can define a set S(T_v, k) as the set of all integers w, called by Pivot Weight Set (PWS), where w is the weight of pivot tree of each near k-[l, u]partition of T_v. S(T_v, k) can be formally presented as follows:

S(T_v, k) = \{w | w = W_v(\bar{P}^k_v(j)) for each \bar{P}^k_v(j) of T_v \}

Here, an element of S(T_v, k) is called fitting if it’s value is in [l, u]. Based on S(T_v, k), we define a compression operation CP_l() for S(T_v, k) and a new set S’(T_v, k) as follows:

S’(T_v, k) = CP_l(S(T_v, k)) = \{w | w ∈ S(T_v, k), w < l\} \cup \{\min\{w | w ∈ S(T_v, k), w ≥ l\}\}

Clearly, the compression operation CP_l() is to obtain a set S’(T_v, k) such that only the minimum fitting element is kept if there have fitting elements in S(T_v, k), by deleting all fitting elements of S(T_v, k) except the minimum fitting one.

Obviously, S(T, p) = \{w | w = W_v(\bar{P}^k_v(j)) for each \bar{P}^k_v(j) of T\}, where r is the root node of T. Then, if S(T, p) contains an element w, l ≤ w ≤ u, T’s p-[l, u]partition is feasible. In other word, T has an p-[l, u]partition if there is a near p-[l, u]partition P^k_r with the weight of it’s pivot tree in [l, u], i.e., W_v(P^k_r) ∈ [l, u].

With the above definitions, we can use a bottom-up approach to combine near k-[l, u]partitions for subtrees to obtain a global solution of near p-[l, u]partitions for the root tree. For example, we assume a subtree T_v has m children v_1, v_2, . . . , v_m from left to right, as illustrated in Fig. 2(a). We first compute the p sets \{S(T_v, k) | k = 1, . . . , p\} for each T_v, and then we arrive at node v and compute the p sets \{S(T_v, k) | k = 1, . . . , p\} by means of dynamic programming similar to [1], [3], [5]. The details of computing S(T_v, k) are described briefly as follows.

For a subtree T_v in Fig. 2(a), with every edge (e, v), we associate a subtree T_v^e defined in the following way: it consists of node v, the i edges (v, v_i), . . . , (v, v_e), the subtree T_v, and all subtrees rooted at left brothers of node v_i, i.e., T_v_{v_1}, T_v_{v_2}, . . . , T_v_{v_{m-1}}. Here, T_v^e = v, T_v^m = T_v. Clearly, T_v^e can be obtained by adding T_v to T_v^{e-1}. Given a near k_1-[l, u]partition of T_v^{e-1} and a near k_2-[l, u]partition T_v, we obtain a near [l, u]partition for T_v, by either adding pivot tree of \bar{P}^k_v to the pivot tree of \bar{P}^k_v or making

Fig. 2 (a) Tree T_v, and (b) If-subtrees after pre-processing to T_v, where l = 5 and u = 14.
pivot tree of $T_{i}$ to form its own subtree as illustrated in Fig. 2(b). Hence, $S(T_{v}, k)$ can be derived from $S(T_{v}, k_1)$ and $S(T_{v-1}, k_2)$, where $k_1$ and $k_2$ range from 1 to $k$, with the condition that $k = k_1 + k_2$ or $k_1 + k_2 - 1$ for different two cases. Therefore, step by step, we can obtain $p$ sets $\{S(T_{v}, k) | k = 1, \ldots, p\}$ based on $\{S(T_{v}, k) | k = 1, \ldots, p\}$, $i \in [1, \ldots, m]$. We will analyze the computing detail in Sect. 4.2.

From property 2, we can derive that the least fitting partition among all $k$-$[l, u]$ partitions for any subtree $T'_{v}$ is always sufficient to examine whether a global tree $T$ has a $p$-$[l, u]$ partition, except for those near $k$-$[l, u]$ partitions with pivot tree’s weight less than $l$. Therefore, in the process of bottom-up traversing and computing, we only need to compute $\{S(T'_{v}, k) | k = 1, \ldots, p\}$ for each node $v$ with $i \in [1, \ldots, m]$, until we arrive at root node. If $S(T, p)$ contains an element $w$ with $l \leq w \leq u$, $T$'s $p$-$[l, u]$ partition is feasible. So we have the following lemma.

Lemma 1. Tree $T$ has a $p$-$[l, u]$ partition if and only if $S(T, p)$ contains an integer $w$ such that $l \leq w \leq u$.

4.2 Basic Algorithm for Minimum $[l, u]$ Partition Problem

Now we present our algorithm to solve the minimum $[l, u]$ partition problem for an arbitrary tree $T$. The pseudocode of the basic algorithm is described in Algorithm 2. By pre-processing sub-algorithm, we firstly translate the original tree $T$ into 1f-trees (line 1 of Algorithm 2). Then we can obtain the globally minimum solution by solving the minimum $[l, u]$ partition problem for each 1f-tree (line 2 to 31). Now we concentrate on minimum $[l, u]$ partition problem for 1f-tree specifically. We traverse each node in post order and compute $p$ sets $\{S'(v, k) | k = 1, \ldots, p\}$ along a 1f-tree $T$, beginning at the leaf node level (level 1) and going up to the root level. Here, $p$ could be any integer larger than the number of subtrees for minimum $[l, u]$ partition. For simplicity, we choose $p = \left\lfloor \frac{m}{2} \right\rfloor$.

- For any node $v$ of level 1 (leaf node level):

  $S'(v, 1) = w(v)$; $S'(v, k) = \phi$ for each $k, 2 \leq k \leq p$. (line 5 to 6)

- For any node $v$ of level 2:

  Here, a node of level 2 refers to a node for which all its children are leaf nodes. Since all children of node $v$ are fitting leaf nodes, we have the following lemma.

Lemma 2. Let $T_v$ be a subtree with an internal node $v$ and $m$ fitting leaf nodes (i.e., $v$ has $m$ children), $V$ be the set of all nodes of $T_v$, $HV(v)$ the set of the heaviest $k$ sons of $v$. Then, (1) $S'(T_v, m + 1) = \{w(v)\}$; (2) for $k = 1, \ldots, m$, we have:

$$S'(T_v, k) = \begin{cases} 
\phi, & \text{if } \sum_{v \in V - HV(k-1)} w(v) > u, \\
\sum_{v \in V - HV(k-1)} w(v), & \text{otherwise.}
\end{cases}$$

Proof. Since $T_v$ has exactly $m + 1$ nodes, there is only one near $(m + 1)$-$[l, u]$ partition, i.e., $P_{T_v}^{m+1} = V$. Hence, $S'(T_v, m + 1) = \{w(v)\}$. Obviously, $S'(T_v, k) = \phi$ for $k > m + 1$. Now we compute $S'(T_v, k), k = 1, \ldots, m$. To get a near $k$-$[l, u]$ partition, we must cut off $k - 1$ leaf nodes from $T_v$, forming a pivot tree having $m - k + 2$ nodes. Clearly, this pivot tree has no less than 2 nodes: $v$ and some leaf nodes. Since any leaf node is fitting, the weight of pivot tree must be larger than $l$. Therefore, if $S(T_v, k)$ has any element, it must be fitting. Obviously, the leastest $k$-$[l, u]$ partition is formed by cutting off the heaviest $k - 1$ leaf nodes, i.e. minimizing $\sum_{v \in V - HV(k-1)} w(v)$, where $V(k-1)$ denotes the set of any $k - 1$ leaf nodes. If $\sum_{v \in V - HV(k-1)} w(v) > u$, then $\sum_{v \in V - HV(k-1)} w(v) \geq \sum_{v \in V - HV(k-1)} w(v) > u, S'(T_v, k) = S(T_v, k) = \phi$. If $\sum_{v \in V - HV(k-1)} w(v) \leq u$, then $S'(T_v, k) = \{w | \ w \in S(T_v, k), w \geq l\} = \sum_{v \in V - HV(k-1)} w(v)$. We thus have the proof. \qed

Based on Lemma 2, we use a simple method to compute $S'(T_v, k)$ (line 7 to 19). We successively cut off k heaviest children of $v$, until $w(T_v) \leq u$. At this phase, clearly, $S'(T_v, k) = \phi, k = 1, \ldots, i, S'(T_v, i + 1) = w(T_v)$. Later, we continue cutting off the heaviest child of $v$ and computing $S'(T_v, i + 2), S'(T_v, i + 3), \ldots$, one by one. The process stops till we arrive at the last child. At last $S'(T_v, m + 1) = w(v)$.

**Algorithm 2 Finding minimum $[l, u]$ partition for a tree $T$.**

**Require:**

- $T$ tree;

**Ensure:**

- minimum partition number $p_{MIN}$;

1. $TS = \text{pre-processing}(T)$;
2. for each 1f-tree $T_j$ of $TS$ do
3. $T = T_j, p = \left\lfloor \frac{m}{2} \right\rfloor$;
4. for all nodes $v$ of $T$ in post order do
5. $S'(T_v, 1) = S(T_v, 1) = w(v)$;
6. $S'(T_v, k) = S(T_v, k) = \phi, 2 \leq k \leq p$;
7. if height($T_v$) = 2 then
8. $i = 1$;
9. while $w(T_v) > u$ do
10. $S'(T_v, i) = \phi$;
11. cut off the child with heaviest weight;
12. $i = i + 1$;
13. end while
14. while $w(T_v) \geq w(o)$ do
15. $S'(T_v, i) = w(T_v)$;
16. cut off the child with heaviest weight;
17. $i = i + 1$;
18. end while
19. $S'(T_v, k) = \phi, i \leq k \leq p$;
20. end if
21. if height($T_v$) > 2 then
22. for $i = 1$ to childcount($v$) do
23. for $k = 1$ to $p$ do
24. compute $S'(T_v, k), S_v(T_v, k)$;
25. $S'(T_v, k) = CP(S_v(T_v, k) \cup S_v(T_v, k))$;
26. end for
27. end for
28. end if
29. end for
30. $p_{MIN} = \min \{1 \leq k \leq p | S'(T, k) \text{ contains a fitting element} \}$;
31. end for
32. $p_{MIN} = \sum p_{MIN}[1]$.
Obviously, $S'(T_v, m + 2) = \cdots = S'(T_v, p) = \phi$.

For any node $v$ of level higher than 2 (line 21 to 27):

We first present a new definition and a new property.

\textbf{Definition 5.} Let $S_a$ and $S_b$ be two sets of integers. We denote by $S_a \oplus S_b$ a set of integers defined as follows:

\[ S_a \oplus S_b = \{a + b \mid a \in S_a, b \in S_b, a + b \leq u\} \]

\textbf{Property 5.} Let $S_a$, $S_b$, $S'_a$, and $S'_b$ be four sets of integers satisfying $S'_a = CP(T_a)$ and $S'_b = CP(T_b)$. Then $CP(S_a \oplus S_b) = CP(S'_a \cup S'_b)$. $CP(S_a \cup S_b) = CP(S'_a \cup S'_b)$.

Based on definition 5 and property 5, we use the following dynamic programming approach to compute $\{S'(T_v, k) \mid k = 1, \ldots, p\}$.

Firstly, we compute $\{S'(T_v, k) \mid k = 1, \ldots, p\}$:

\[ S'(T_v, 1) = w_0; S'(T_v, 0, k) = \phi \]

Then, we compute $\{S'(T_v, k) \mid k = 1, \ldots, p, i = 1, \ldots, m, \}$ by merging a $S'(T_v, k)$ with $S'(T_i, k)$, where $k = k_1 + k_2 - 1$ or $k_1 + k_2$ for different two cases:

\textbf{case 1: Pivot tree of $P_{T_v}^k$ merging with pivot tree of $P_{T_i}^{k_1}$, $k_1 + k_2 = k + 1$.}

\[ S_1(T_v, k) = \bigcup_{k_1=1} S'(T_v, k_1) \cup S'(T_i, k + 1 - k_1) \]

\textbf{case 2: Pivot tree of $P_{T_v}^k$ staying alone as an independent subtree, $k_1 + k_2 = k$.}

\[ S_2(T_v, k) = \bigcup_{k_1=1} S'(T_v, k_1) \]

where the union is taken over all integers $k_1$, $1 \leq k_1 \leq k - 1$, such that $S'(T_v, k - k_1)$ has a fitting element.

From the two sets $S_1(T_v, k)$ and $S_2(T_v, k)$ above, one can compute $S'(T_v, k)$ as follows:

\[ S'(T_v, k) = CP_a(S_1(T_v, k) \cup S_2(T_v, k)) \]

In the dynamic programming formulation above, the set $S'(T_v, k)$ is computed bottom-up and from left-to-right, $\{S'(T_v, k) \mid k = 1, \ldots, p\}$ have already been treated when we compute $\{S'(T_i, k) \mid k = 1, \ldots, p\}$.

So $\{S'(T_v, k) \mid k = 1, \ldots, p\}$ can be derived level by level, until we arrive at root node $r$, where we can compute $\{S'(T_v, k) \mid k = 1, \ldots, p\}$. Then, one can compute the minimum number $p_{\text{min}}$ for $P_{T_v}^k$, as follows:

\[ p_{\text{min}} = \min\{1 \leq k \leq p \mid S'(T_v, k) \text{ contains a fitting element} \} \]

In this basic algorithm, a simple optimization method is adopted to compute $A \oplus B$ for two sets $A$ and $B$. We start from the level $1$ of $A$ (denoted by $A[1]$) and compute the sum of $A[1]$ and $B[i]$ one by one, $1 \leq i \leq |B|$. Once $A[1] + B[i] \geq l$, we stop the computing of $A[1] + B[i+1]$, and come to $A[2]$ for new iteration, until we finish the computing at the last element of $A$.

It is easy to specify the time complexity of this algorithm: Any $S'(T_v, k)$ has at most one element larger than $l$, hence $|S'(T_v, k)| \leq l + 1 = O(l)$. The $\oplus$ of calculating two sets can be executed in time $O(l^2)$. Then, one can compute the set $S_1(T_v, k)$ and the set $S_2(T_v, k)$ in time $O(p^2)$ and $O(p)$, respectively. Therefore the set $S'(T_v, k)$ is computed in time $O(p^2)$. Last, one can recursively compute the set $S'(T_v, k)$ for $n$ nodes of $T$ in time $O(n^2)$, since there are at most $2n - 1$ pairs of a node $v$ and an integer $i$ in $T$. Thus the basic algorithm has a worst-case runtime of $O(p^2 n)$.

Obviously, Algorithm 2 can be slightly modified to actually obtain the $P_{T_v}^{k_{\text{min}}}$ which tells the specific positions of partitioning. As we know, each $w \in S^1(T_v, k)$ corresponds to a $P_{T_v}^k$. If $w \in S^1(T_v, k)$, the edge $(v, w)$ is not deleted for $P_{T_v}^k$; then we attach a tuple $(1, w, k')$ to the $w$, where $w = w + w_1, w_1 \in S^1(T_{v+1}, k')$ and $w_2 \in S_2(T_v, k - k' + 1)$. If $w \in S^2(T_v, k)$, the edge $(v, w)$ is deleted for $P_{T_v}^k$; we attach a tuple $(2, k')$ to the $w$, where $w \in S^2(T_{v+1}, k')$. Using these tuples, one can use a backtracking process to identify those nodes whose parent edges are deleted in the minimum $[l, u]$ partition of $T$. By this way, we actually find the minimum $[l, u]$ partition.

### 4.3 Algorithm Optimization

From Sect.4.2, we know the time complexity of Algorithm 2 is affected by the size of dynamic programming set $S'(T_v, k)$. An opportunity for algorithm optimization is to simplify the representation and computation of $S'(T_v, k)$. An approach used in [3] is adopted here. The main idea is to transform $S'(T_v, k)$ to a set of intervals (i.e., a pair of integers), where each interval represents a “maximal d-consecutive subset” derived from $S'(T_v, k)$. For the soundness of this paper, we present the optimization approach briefly, omitting the proof of algorithm optimization. One can refer to [3] for relative theoretical details. Now we first present some definitions.

An ordered set of integers is called d-consecutive if the value difference of any two consecutive elements is at most $d$, where $d = u - l$. For a non-empty set $A$ of integers and a non-empty subset $A' \subseteq A$, $A'$ is said to be a maximal d-consecutive subset of $A$ if $A'$ is d-consecutive and $A' \subseteq A$ does’t hold for any other d-consecutive subset $A''$ of $A$.

Now we define a set of intervals $I(A)$, based on a set of integers $A$:

\[ I(A) = \{[a, a'] \mid [a, a'] \text{ is the interval of a maximal d-consecutive subset of } A \} \]

For two sets of intervals $I(A)$ and $I(B)$, we define $I(A) \oplus I(B)$ as a set of intervals as follows:

\[ I(A) \oplus I(B) = \{[a + b, a' + b'] \mid [a, a'] \in I(A), [b, b'] \in I(B), a + b \leq u \} \]

Without loss of generality we assume four integers $a, a', b, b'$, with $a \leq b, b' > a, b' > b$. The interval $[a, a']$ is said to be intersected with interval $[b, b']$ if $b - a' \leq 0$. The interval $[a, a']$ is said to be d-interfering with $[b, b']$ if $b - a' < d$. We define a merge operation $\cup$ for two $d$-interfering intervals as $[a, a'] \cup [b, b'] = [a, \max(a', b')]$.

For a set of intervals $I$, if we repeatedly merge $d$-interfering intervals of $I$ until no any two intervals are $d$-interfering, then we can get a new set with no intervals $d$-interfering. We denote the operation to set $I$ by $M(I)$ and the new set by $M(I)$.

From the above definitions, one can have following properties:
Property 6. Let $\tilde{A}$ and $\tilde{B}$ be two $d$-consecutive integer sets with all elements no more than $u$, and let $a_{\text{min}}$ ($b_{\text{max}}$) be the minimum and maximum elements of $\tilde{A}$ ($\tilde{B}$) respectively. If the interval $[a_{\text{min}}+b_{\text{min}},a_{\text{max}}+b_{\text{max}}]$ is intersected with $[l,u]$, there must exist an element $a \in \tilde{A}$ and an element $b \in \tilde{B}$ such that $l \leq a+b \leq u$, and vice versa.

Property 7. Let $I$ be a set of intervals with some intervals $d$-interfering with each other. There must exist an interval $M(I)$ intersected with $[l,u]$, if an interval of $I$ intersects with $[l,u]$.

With the above definitions and properties, the optimized algorithm computes the set $I(S'(T_v,k))$ in place of $S'(T_v,k)$. With the same approach of dynamic programming as Algorithm 2, we recursively compute interval sets $I(S'(T_v^{k-1}), k) = [1, \ldots, p]$ for each node $v$ of $T$ from the leaves to the root, and hence computes $I(S'(T,k)) = [1, \ldots, p]$. We denote $I(S'(T_v^{k-1}), k)$ by $I(T_v^{k-1}, k)$, one can have the following lemma:

Lemma 3. Tree $T$ has a $p$-interval partition if and only if $I(T, p)$ contains an interval $[a, a']$ such that $[a, a']$ intersects with $[l,u]$.

Hence, we can recursively compute interval sets $I(T_v^{k-1}, k)$ for $k = 1, \ldots, p$ as follows. For any node $v$ of level 1:

- $I(v, 1) = \{w(v), w(v)\}$; $I(v, k) = \emptyset$, $2 \leq k \leq p$.
- For any node $v$ of level 2:
  - Each $T_v(k)$ has at most one element, therefore, for $1 \leq k \leq p$,
    
    $$I(T_v, k) = \begin{cases} \{|w, w| w \in S'(T_v(k))\} & S'(T_v, k) \neq \emptyset, \\ \emptyset & \text{otherwise.} \end{cases}$$
  - For any node $v$ of level higher than 2:
    Firstly, $I(T_v^{k-1}, k) = \{w(v), \} w(v)\}$; $I(T_v^0, k) = \emptyset$, $2 \leq k \leq p$.

Then, we compute the $p$ sets $I(T_v^{k-1}, k)$ incrementally, for $i = 1$ to $m$, distinguishing two cases:

- case 1: Two pivot trees merging together
  
  \[ I_1(T_v^{k-1}, k) = \bigcup_{k_i=1}^{k} I(T_v^{k-1}, k_i) \cup I(T_v^{k-1}, k-1+1). \]

- case 2: Pivot tree of $T_{\phi}$ staying as independent subtree
  
  \[ I_2(T_v^{k-1}, k) = \bigcup_{k_i=1}^{k} I(T_v^{k-1}, k_i), \]

where the union is taken over all $k_i$, $1 \leq k_i \leq k-1$, such that $I(T_v^{k_i}, k_i)$ has an interval intersected with $[l,u]$. By unifying the above two sets and doing merge operation, we have:

\[ I(T_v^{k}, k) = M(I_1(T_v^{k-1}, k) \cup I_2(T_v^{k-1}, k)), \]

By the above dynamic programming, whether $T$ has a $p$-interval partition can be checked from $I(T, p)$. By adopting a method like Sect.4.2 (see also [3] for the detail), one can actually obtain $P_T^{\text{opt}}$.

We now turn to the runtime complexity of the optimized algorithm. Any $I(T_v^{k}, k)$ has at most $\frac{u-a}{d+2} + 1$ elements, hence $|I(T_v^{k}, k)| = O(N)$. One can compute the \( \bigcap \) of any two interval sets in time $O(\frac{u-a}{d+2})$. Then, $I_1(T_v^{k}, k)$ and $I_2(T_v^{k}, k)$ can be computed in time $O(\frac{u-a}{d+2})$ and $O(\frac{u-a}{d+2})$ respectively. Therefore one can compute the set $I(T_v^{k}, k)$ in time $O(\frac{u-a}{d+2})$. Lastly, we can recursively compute the $p$ sets $I(T_v^{k}, k)$ for $k = 1, \ldots, p$ for $n$ nodes of $T$ in time $O(n)$. Thus the basic algorithm has a worst-case runtime of $O(n^3)$. The optimized algorithm can be slightly modified to actually obtain a $P_T^{\text{opt}}$, analogously as the method in Sect.4.2.

5. $p$-interval partition problem for arbitrary tree

We only need to modify our aforementioned algorithms slightly, to solve the $p$-interval partition problem for arbitrary tree. Similarly, we first translate tree $T$ into $lf$-trees. We assume $T$ is translated into $N$ lf-trees after preprocessing. For each lf-tree denoted by $T(i)$, we compute and obtain $I(T(i), k)$ for $k = 1, \ldots, p$ by Algorithm 2. Then the $p$-interval partition problem can be presented as follows: given $I(T(i), k)$ for $k = 1, \ldots, p$ for each lf-tree $T(i)$, find an integer $k_i$ and corresponding $k_i$-[l,u]partition $P_{T(i)}^{k_i}$ for each $T(i)$, such that $\sum_{i=1}^{N} k_i = p$. We use a simple dynamic programming scheme to get the solution, as follows.

For any $T(i)$, we define a set $Y_i$ of size $p$, whose element is computed as follows:

\[ Y_i(k) = \begin{cases} 1 & I(T(i), k) \text{ contains an interval intersected with } [l,u], \\ 0 & \text{otherwise}. \end{cases} \]

We denote a set of lf-trees $(T(1), \ldots, T(i))$ by $T(1 \rightarrow i)$. Based on $T(1 \rightarrow i)$, we define a set $Y_1$ of size $p$, whose element is computed as follows:

\[ Y_1(k) = \bigcup_{k_i=1}^{k-1} (Y_1 \cup Y_i(k-1)) \]

where \&\& indicates logical AND operator and \bigcup \bigcup \bigcup indicates logical OR operator.

We incrementally compute the set $Y_i$ from $i = 1$ to $N$, until we get $Y_N$. If $Y_N[p] = 1$, $T$’s $p$-[l,u]partition is feasible. Then, we can obtain $k_i$ and $k_i$-[l,u]partition $P_{T(i)}^{k_i}$ for each $T(i)$ satisfying $\sum_{i=1}^{N} k_i = p$, through a backtracking process of iterative computing. Then $\bigcup_{i=1}^{N} P_{T(i)}^{k_i}$ is our solution.

6. Experimental Results

In this section, we evaluate the performance of our algorithm. In the simulations, trees are synthetically generated, specified mainly by three parameters: the tree depth (denoted by treeDepth), the range of a node’s degree (denoted by degreeRange), and the range of a node’s weight (denoted by weightRange). The number of nodes for a generated tree is called the size of the tree. We compare the performance of our algorithm against the algorithm by Ito et al. [3] in executing time. These algorithms are coded in C++ programming language and executed on a Pentium IV PC with 2.6 GHz CPU.
6.1 Impact of Pre-processing Algorithm on Tree Size

In this experiment, we compare the size of original trees with those after pre-processing. We randomly generate a group of trees, each of which is generated with treeDepth = 6, weightRange = [1, 10], degreeRange = [2, 6]. Firstly, we choose the fixed value of \([l, u]\) for these trees, i.e., \([l, u] = [20, 80]\). Figure 3(a) shows the size of 10 trees before and after pre-processing. Secondly, we set a constant value of \(d\) and choose different \(l\) for these trees. Here, \(d = 60\), \(l\) is varied from 20 to 180. Figure 3(b) shows the size of these trees after pre-processing with different \([l, u]\). It is clear that pre-processing algorithm always decreases the size of a tree, as more significant as \(l\) is larger compared with the average weight of each node. Note that we also generated other groups of trees with different parameters for the above experiments and got the same comparison results as Fig. 3.

6.2 Executing Time of Finding Minimum \([l, u]\)Partition

Here, we compare the performance of our algorithm against algorithm by Ito et al. in executing time. Firstly, we use the same trees generated in Sect. 6.1, and \([l, u]\) is set as \([120, 180]\) for these trees. We run both algorithms to find minimum \([l, u]\)partition of these trees. It’s clear that our algorithm works much faster than the algorithm by Ito et al. Secondly, We randomly generate two new trees \(T(A)\) and \(T(B)\) with the same parameters as Sect. 6.1. The size of the two trees are 1265 and 1810 respectively. We run algorithms of finding minimum \([l, u]\)partition for \(T(A)\) and \(T(B)\). Here \(d\) is set as a constant value \((d = 60)\) and \(l\) is varied from 80 to 180. Figure 4(b) shows the executing time of both algorithms with different value of \(l\). We can see that the running time for both algorithms decreases along with the increasing of the value of \(l\). Obviously, our algorithm works much faster than algorithm by Ito et al. in any case. Note that we also compare our algorithm against algorithm by Ito et al. for the executing time of finding \(p-[l, u]\)partition, we have the same comparison results.

7. Conclusions

We propose new algorithms of solving minimum \([l, u]\) partition and \(p-[l, u]\)partition problems of a tree. To reduce the executing time of the algorithms, we adopt a pre-processing approach to cut down the scale of the original tree, and use a few pruning methods to speed up the computing process. Experiments show that our algorithm work efficiently and outperform the algorithm proposed by Ito et al. greatly.

Acknowledgments

This work was supported by Program for New Century Excellent Talents in University under Grant no. NCET-10-0298, and Foundation of Science & Technology Department of Sichuan Province under Grant No. 2011HH0037.

References

Guangchun Luo received the Ph.D. degree in computer science from University of Electronic Science and Technology of China, Chengdu, China, in 2004. He is currently a professor of computer science at UESTC. His research interests include computer networks, mobile networks, and network security.

Hao Chen received the B.S. (1996) in radio navigation and the M.S. (1999) in signal and information processing from Air force Engineering University, China. He was a research engineer at air force command of China from 2002 to 2012. He is currently a Ph.D. candidate at University of Electronic Science and Technology of China (UESTC). His research interests include data stream, distributed and cloud computing, multisensor data fusion, command automation and data management.

Caihui Qu received the B.S. (2009) in applied mathematics from Chengdu University of Technology, China. He is currently a master candidate at University of Electronic Science and Technology of China. His research interests include data stream, distributed computing and software engineering.

Yuhai Liu received his Ph.D. degree in command automation from the PLA Academy of Ordnance Engineering of China in 2001. He is currently a senior engineer at air force command automation station of China, where he participated in developing many large-scale information systems. His research interests include command automation and information processing.

Ke Qin received his M.E. and Ph.D. degree from the University of Electronic Science and Technology of China in 2006 and 2010 respectively. He was also a visiting scholar at Carleton University in Ottawa, Canada in 2008. His research interests include chaos theory, chaotic neural networks and encryption.